NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

COMMON DATA SHARING SYSTEM
INFRASTRUCTURE: AN
OBJECT-ORIENTED APPROACH

by

Calvin D. Slocumb

June 1995

Thesis Advisor: Orin Marvel

Approved for public release; distribution is unlimited

19960221 053

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 1995	Master's Thesis

4. TITLE AND SUBTITLE

COMMON DATA SHARING SYSTEM INFRASTRUCTURE:
AN OBJECT-ORIENTED APPROACH

5. FUNDING NUMBERS

6. AUTHOR(S)

Slocumb, Calvin D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution is Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

To solve today's problem of the lacking interoperability between disparate command and control (C^2) systems, the Ballistic Missile Defense Office is developing a proof-of-concept model. The proof-of-concept model, Common Data Sharing System Infrastructure, allows external command and control systems to store, transfer, and communicate using a common object-oriented database. The Generation One architecture demonstrates that information could be shared in a client-server architecture using a message-based, flat-file means of transferring data. Generation Two uses an object-oriented database to increase the infrastructure's robustness and flexibility to give information on demand. This thesis evaluates the Common Data Sharing System Infrastructure for its potential use in future command and control systems.

14. SUBJECT TERMS

Object-oriented, interoperability

15. NUMBER OF PAGES

84

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

COMMON DATA SHARING SYSTEM INFRASTRUCTURE:
AN OBJECT-ORIENTED APPROACH

Calvin D. Slocumb
Lieutenant, United States Navy
B.A., University of Memphis, 1989

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN SYSTEMS TECHNOLOGY
(JOINT COMMAND, CONTROL, COMMUNICATIONS, COMPUTERS, AND
INTELLIGENCE SYSTEMS)

from the

NAVAL POSTGRADUATE SCHOOL
June 1995

Author: _____
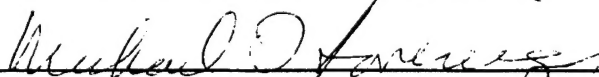   Calvin D. Slocumb

Approved by: _____
        Professor Orin Marvel, Thesis Advisor

_____
        Professor Michael Sovereign, Second Reader

_____
        Professor Dan C. Boger Academic Group Chairman,
Joint Command, Control, Communications, Computers, and Intelligence Systems Curriculum

iii

iv

ABSTRACT

To solve today's problem of the lacking interoperability between disparate command and control (C^2) systems, the Ballistic Missile Defense Office is developing a proof-of-concept model. The proof-of-concept model, Common Data Sharing System Infrastructure, allows external command and control systems to store, transfer, and communicate using a common object-oriented database. The Generation One architecture demonstrates that information could be shared in a client-server architecture using a message-based, flat-file means of transferring data. Generation Two uses an object-oriented database to increase the infrastructure's robustness and flexibility to give information on demand. This thesis evaluates the Common Data Sharing System Infrastructure for its potential use in future command and control systems.

TABLE OF CONTENTS

LIST OF TABLES

xi

x

LIST OF FIGURES

ACKNOWLEDGEMENT

EXECUTIVE SUMMARY

This thesis addresses an issue that continues to plague the development of military command and control systems non-interoperability. The lack of interoperability is the result of military command and control systems that were designed to meet a service-specific need. It is a problem that constantly encumbers battlefield commanders in warfare. The Persian Gulf War is the most recent example of the lack of inter-operability with command and control assets between the different military services and within the same service. This problem is evident also between different nations which operate together in coalition forces. This thesis examines the use of a software and hardware solution to this problem of non-interoperability.

A subsidiary organization from the Ballistic Missile Defense Office at the National Test facility in Colorado Springs, Colorado is pioneering the development of an object-oriented, proof-of-concept infrastructure. Through its development and testing at a series of demonstrations, such as the Joint Warrior's Interoperability Demonstration 1993 and 1994, this proof-of-concept demonstrates its ability to provide seamless interoperability between disparate command and control systems. This proof-of-concept model is called Common Data Sharing System Infrastructure (CDSSI). CDSSI is being developed by a team of experts at the Battlespace Management Command, Control, and Communications (BM/C^3) Element Support Center.

BESC shows through a series of demonstrations with a Generation One model that interoperability is achievable by using commercially available technology. The Generation One version of CDSSI also demonstrates that the use of flat files for persistent information storage is not the most effective means to achieve inter-operability. Generation One testing reveals that message transfers using a fixed and variable lengthed data structure reduces the infrastructure's flexibilty to adapt to changing requirements from external systems. Consequently, the service given to the external systems is inadequate to meet the system's functional requirements.

Object-oriented development techniques provided the flexibility and robustness needed to meet the system's requirements. These techniques are applied to a Generation Two proof-of-concept infrastructure. The long range development activities for the Common Data Sharing System Infrastructure provide a framework for its daily development activities. The Common Data Sharing System Infrastructure develops in "build a little, test a little" fashion. The proof-of-concept uses a commercially available database management software tool called, ObjectStore, to give the infrastructure the flexibility needed.

Object-oriented analysis, design, and programming reduces the complexity in designing large systems. There are, of course, no "silver-bullet" solutions to software or hardware problems. There are available tools and methodologies that have proven themselves to be effective in successfully building interoperable systems. Many of the techniques have been practiced in the past, like distributed computing. Commercially, new object-oriented application building tools and new programming languages are emerging. Still older methods are evolving.

The Common Data Sharing System has demonstrated its effectiveness at several demonstrations, including the Joint Warrior Interoperabiliy Demonstrations with the participation of sites throughout the country. It deserves considerable attention, simply because the infrastructure is applicable to solving the warrior's interoperability problem. It also uses proven techniques for allowing interoperability to be achieved. At this point, much of the information about the Common Data Sharing System is not available because some proprietary regulations which prevent the sharing of data to public sources. Nonetheless, what is known is that the Common Data Sharing System Infrastructure has the potential to improve a battlefield commander's ability to seamlessly obtain information and share it on demand or when he needs it most. This research hopes to lead the way to solving one of command and control's most plaguing issues for battlefield commanders -- interoperability.

I. INTRODUCTION TO COMMON DATA SHARING SYSTEM INFRASTRUCTURE

A.   BACKGROUND

The need for interoperability between command and control (C^2) systems bears considerable study. It is not an easily solvable problem, yet the objective of this thesis is to research a possible solution to the C^2 interoperability problem using object-oriented techniques.

Chapter I discusses the general idea behind a proof-of-concept infrastructure called Common Data Sharing System Infrastructure. Chapter II discusses a new systems development paradigm which uses object-oriented techniques to decompose complex systems and reconstruct more simplified ones using this methodology. It also discusses the basic components of object-oriented techniques, objects and classes. Chapters III and IV apply object-oriented principles to the development of the Common Data Sharing System Infrastructure. Chapter V describes how the infrastructure is applied to othe command and control systems. Chapter VI discusses the design of object-oriented interfaces. Chapter VII addresses some the problems discovered during the design and testing of the infrastructure. Chapter VIII provides a discussion of some of the current object-oriented technologies available. Finally, Chapter IX gives an interpretation of some findings during development and summarizes the previous discussions.

B.   INTEROPERABILITY

One of the common problems that systems engineers and technologists seek to solve in the design of command and control systems is that of interoperability. Traditionally, the Army, Air Force, Navy, and Marine Corps have developed command and control (C^2) systems, commonly called "stovepipes". Now, the military services can no longer acquire stovepipe command and control systems. This paradigm shift is due largely to Department of Defense mandates and budget reductions. More importantly, this shift is driven by the already grown complexity of large software systems, the

1

widespread acceptance of commercial standards, and the lack of interoperability during battlefield scenarios.

C. INFORMATION WARFARE

Technology in developing command and control systems is also leading to advanced technologies in information control on the battlefield. This new area of battlefield information technology is broadly defined as *information warfare.* As command and control systems become more complex, simplified ways to pass information between these systems must be found. Thus, if information warfare is another piece of the battlespace management puzzle, then the commander must have a way to integrate this information into a common tactical picture.

D. CDSSI

The Battlespace Management/Command, Control, and Communications (BM/C^3) Element Support Center, also known as *BESC*, is a program which began within the National Test Facility in 1992. BESC is a part of the larger Ballistic Missile Defense Organization (BMDO). Its mission is to explore and use new methodologies to build and integrate prototypical BM/C^3 systems. BESC is using an efficient software design methodology to create a proof-of-concept command and control support system called *Common Data Sharing System Infrastructure* (CDSSI). CDSSI is BESC's solution to bridge the gap between disparate, non-interoperable C^2 systems. CDSSI provides information transfer, data conversion, system storage, monitoring, process control, and interprocess communications between the different external hardware and software systems connected to CDSSI. [Ref. 1]

1. Goals

As described by Don Ravenscroft, a developer of CDSSI at BESC, the primary goal of CDSSI is to apply the new development methodology to build a proof-of-concept infrastructure that would provide seamless connectivity between disparate C^2 systems [Ref. 2]. Secondary goals were:

2

- Using the information architecture (IA) as the initial System Description, use object-oriented design to define the CDSSI to be robust enough to easily adapt to changing requirements and design changes.

- To implement a prototype CDSSI based on these IA definitions using object-oriented tools and an object-oriented database.

- Investigate the performance of the CDSSI to support real C^2 systems.

- To use the prototype CDSSI in demonstating interoperability among C systems in real-world demonstrations using both real and simulated C^2 systems in a realistic environment.

2. IP and CDSS

Figure 1 shows the configuration setup for CDSSI in a client-server relationship. The Common Data Sharing System Infrastructure is composed of two main parts: the interprocess communications (IP) and the Common Data Sharing System (CDSS). Using the client-server approach for information transfer, the initial IP function provided connectivity between the CDSS and other outside systems. These outside systems are referred to as external clients. In addition, the IP provided information storage within CDSSI by messages and flat files.

The CDSS, however, is the key to the infrastructure. The Common Data Sharing System provides the seamlessness in connecting the dissimilar C^2 systems. Current

issues, regarding information security, information access authorization, interpretation of common information, and distribution techniques are being explored more fully as the
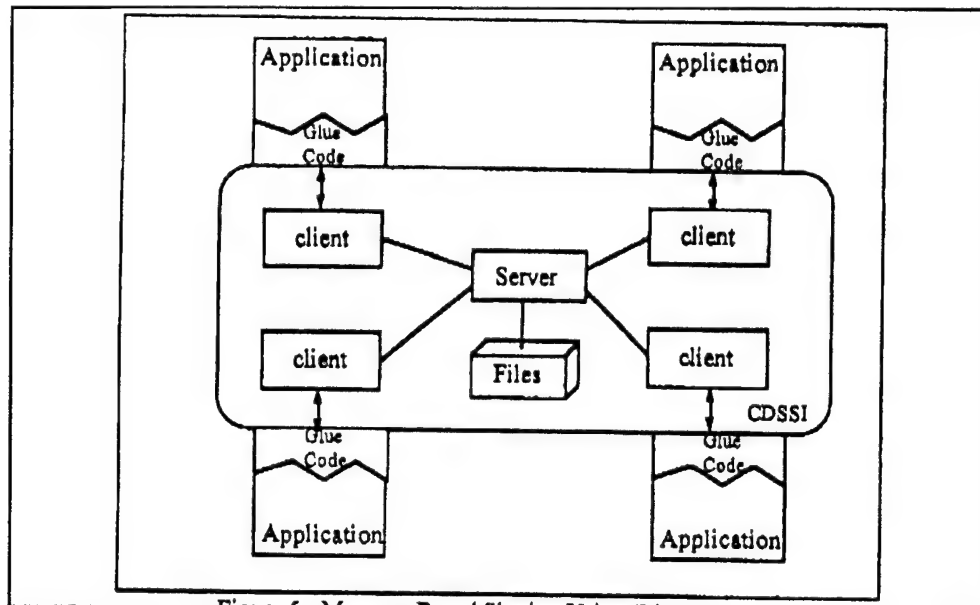
Figure 1. CDSSI as a Client-Server Architecture. From Ref. [2].

CDSS moves from the proof-of-concept stage to the deployment stage. The basic design of the architecture encompasses using CDSS to handle the function of interoperability-providing available information to different systems.

3. Methodology

The CDSSI design methodology involves rapid prototyping to incrementally develop the system at four levels of maturity: Information Architecture Build, Design Architecture Build, First Deployment Build, and Second Deployment Build. The levels of maturity are based upon the BM/C^3 Directorate's new design methods to build and test new military C^2 systems within the Department of Defense. Figure 2 depicts these levels of maturity.

4

This design methodology involves linking real and simulated systems together to demonstrate the CDSSI's ability to integrate operational systems in a real environment

Prototype • Incremental Builds

Second Deployment Build

First Deployment Build

• • •

Design Architecture Build

Information Architecture Build

Maturity

Time
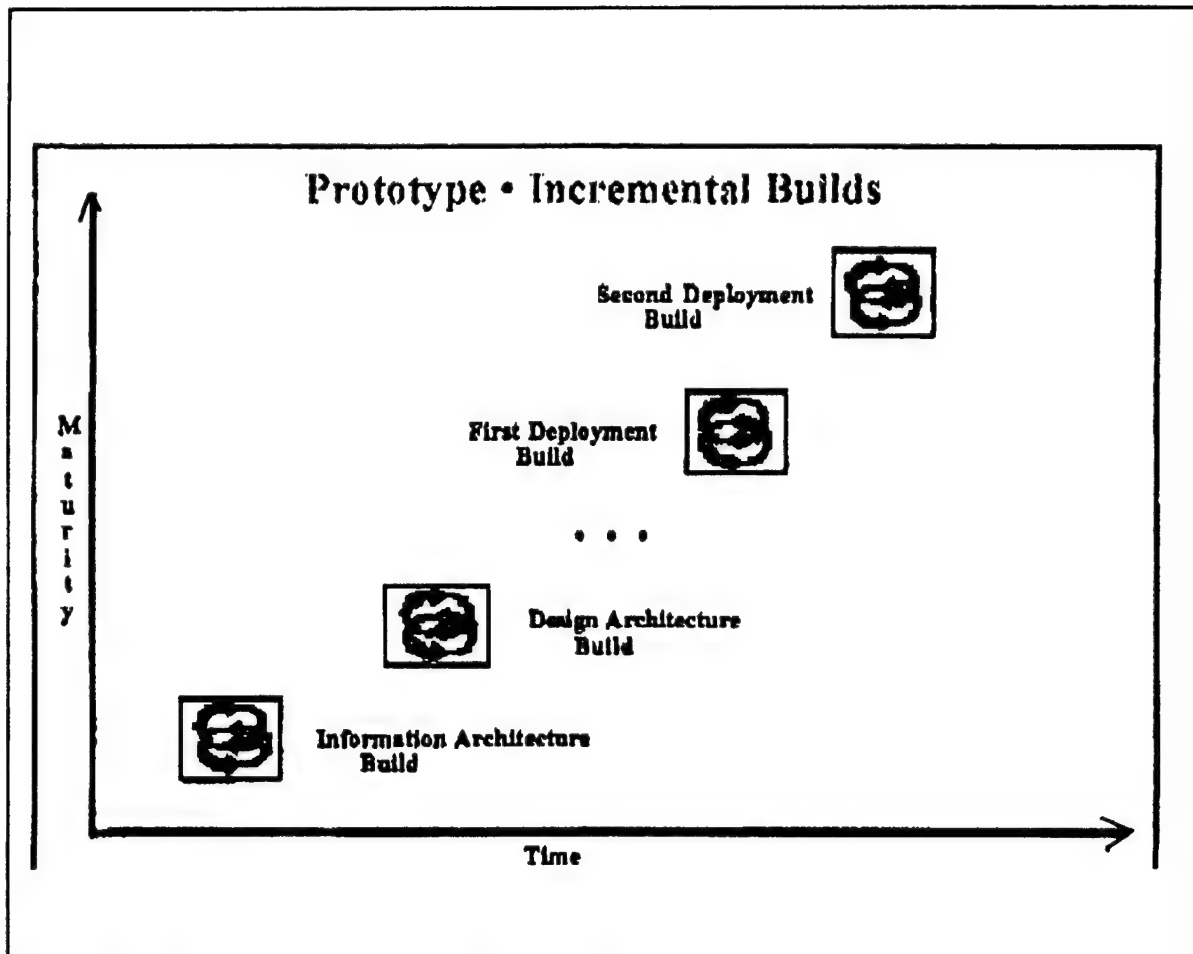
Figure 2. Deployment Methodology. From Ref. [2].

against a simulated ballistic missile threat. Ravenscroft states that the methodology's principles are to:

- Use an Information Architecture to capture system behavior and characteristics.

- Use rapid prototyping to provide incremental builds for use and evaluation.

5

- Use best commercial software development practices in developing new software.

- Use commercial products and standards whenever possible.

- Maximize legacy or re-use software applications whenever possible.

- Encourage innovation and programmer productivity. [Ref. 2].

The methodology is based upon object-oriented analysis (OOA) and object-oriented design (OOD). Chapter II will discuss these concepts in further detail. It is suffice to say, however, that OOA and OOD techniques offer significant advantages to building a common infrastructure for C^2 systems. Primarily, the advantages of OOA and OOD are the ability to control redundancy and share common data among users. [Ref.3].

Therefore, CDSSI offers a viable opportunity to enhance the capabillities of a single C^2 system, like the Navy's Joint Maritime Command Information System (JMCIS), by integrating it into a common user database for both local and remote C^2 systems. CDSSI offers the robustness and flexibility needed to provide seamless information sharing.

4. Requirements

To begin the development of any system, one must generate requirements to guide the system's progress. The requirements for CDSSI stem from the infrastructure's underlying goal to provide seamless connectivity between disparate systems. The requirements attempt to capture behavior characteristics, such as robustness and flexibility, which are inherent in CDSSI. The requirements are as follows:

- The infrastructure must be able to interface with diverse external existing systems both real and simulated without requiring changes to the external systems. The infrastructure must be able to transfer information to/from any external system.

- The infrastructure must provide a consistent information access mechanism to all using systems.

- Persistent information shall be stored by the infrastructure.

6

- The infrastructure shall be capable of performing in real-time. (Real-time, in this context, is defined as having no delay caused by the infrastructure systems that is unacceptable to any system.)

- The infrastructure should be able to easily adapt to new systems. Adding new systems should require creation of minimal *glue code* interfacing software.

- The infrastructure should be designed and implemented using open systems techniques. [Ref. 4].

It is important to note that these requirements for CDSSI are generated based upon the inputs from the BESC team and external C2 systems users. This is because the BESC system engineers have the resources (i.e., analysts, managers, testers, quality assurance specialist, designers, programmers, and users) while the external users (Joint Task Force headquarters, simulated Joint Forces Air Component Command headquarters, and other sea, air, and land component command headquarters) know what is needed and how the system is suppose to behave. It is also worthy to note that CDSSI was developed in two Generations which will be discussed in further detail in Chapter V. The process for developing CDSSI was iterative in that the project was divided into smaller tasks. These tasks were then analyzed, designed, developed and tested with the users' input throughout the process. Some requirements changed as the system evolved toward the Second Deployment Build shown in Figure 2.

E. GENERATION ONE DEVELOPMENT

Generation One development of CDSSI focuses upon the Common Data Sharing System (CDSS). This is the first phase in which CDSSI was demonstrated and tested with real and simulated C^2 systems. Generation One development also gives CDSSI the exposure on a large scale to demonstrate not only the viability of the system, but also the viability of the process.

The demonstration was conducted in 1993 through a series of live demonstrations which ended in a large demonstration in October 1993. The design and analysis of CDSSI was based upon the standard functional analysis approach to

systems engineering. As with most tests, there were some design flaws discovered with the Generation One version of CDSSI that will be discussed later in this thesis. Nonetheless, the demonstration was considered successful. The lessons learned from Generation One were used as a springboard for the development of Generation Two architecture.

F. GENERATION TWO DEVELOPMENT

Generation Two development of CDSSI was the follow-on build to Generation One. As mentioned earlier, the approach was to capitalize on the lessons learned from the previous development. Since the Generation One architecture required improvement, the new architecture gave CDSSI more robustness and flexibility by using another approach to systems engineering. This approach uses the advantages of object-oriented technologies in analysis, design, and programming to give CDSSI an innovative edge in development.

The proof-of-concepts that are discussed attempt to integrate disparate command and control systems. Object-oriented techniques will be discussed in Chapter II to explain their value in developing systems.

II. WHAT IS OBJECT-ORIENTED?

A. OBJECT-ORIENTED

This chapter discusses what object-oriented means. It also provides a definition of some basic terminology related for object-oriented systems development. The terms discussed in this chapter are the basics to understanding the methodology for the Common Data Sharing System Infrastructure.

B. ALGORITHM DECOMPOSITION

In order to understand object-oriented techniques of decomposing and building systems, one should understand the more traditional approach to system design. The latter approach is referred to as algorithm or functional decomposition.

Systems are decomposed to allow humans to deal with the complexity of whole systems. Psychologist G. Miller stated that humans are only able to process a maximum of seven "chunks" of information, plus or minus two, at a time. [Ref. 5] As humans attempt to understand larger, more complex systems, algorithm decomposition provides a way to break down the system into more manageable pieces. The system is broken down in a top-down approach into hierachical levels. Figure 3 below describes this approach.

Decomposition by using the structured algorithm approach has strengths and limitations. The advantage to algorithm decomposition is that it provides a logical view to the ordering of events. This structured approach falls apart when large software programs exceed 100,000 lines of code, or so. [Ref. 6] Algorithm decomposition is also limited in that it doesn't address the issues of data abstraction, encapsulation, or concurrency, which will be discussed further.
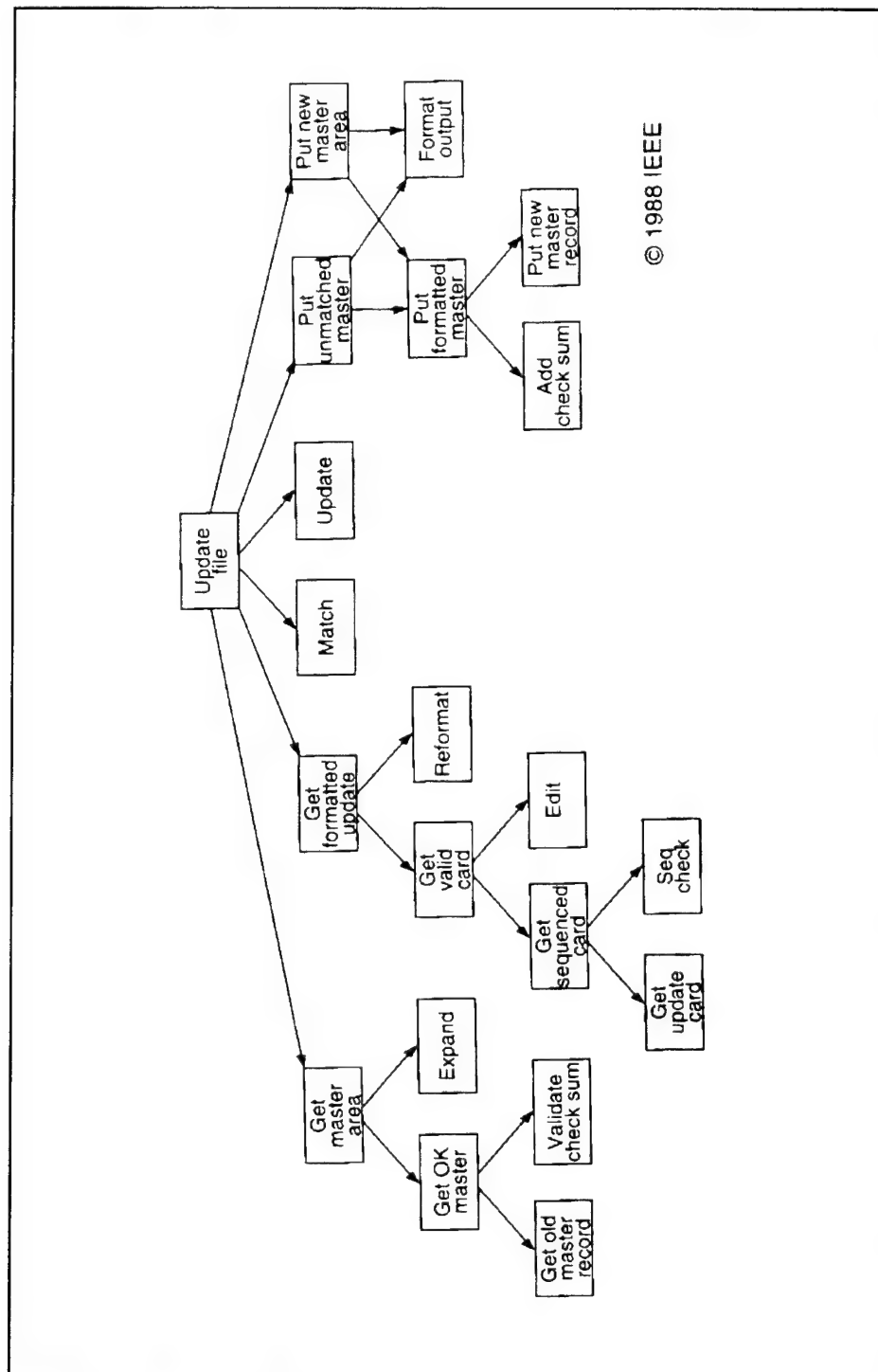
Figure 3. Algorithm Decomposition. From Ref. [8].

C. OBJECT-ORIENTED DECOMPOSITION

In contrast to algorithm decomposition, object-oriented decomposition provides a manageable way to decompose and understand large, complex systems. Object-oriented decomposition models the way humans deal with complex information in that the problem domain is broken into objects with uniquely identified behaviors and attributes. These objects then interact by sending messages to one another. Figure 4 describes the object-oriented approach to decomposition.

Some complex systems, like command and control systems, have the terms, C^2, C^3, C^3I, C^4I, C^4I^2, etc., associated with the system. These context specific terms tend to be overused, and oftentimes misused, but generally refer to the same thing. Likewise, the term, *object-oriented,* is overused and misused. Rentsch's comments are as accurate today as when he stated them over ten years ago, when he said,

> "My guess is that object-oriented...will be in the 1980s what structured programming was in the 1970s. Everyone will be in favor of it. Every manufacturer will promote his products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know what it is."[Ref. 7].

1. The Object Model

It is fair to say that the concept of *object* is central to anything labeled object-oriented. An object is a tangible entity that exhibits some well-defined behavior. [Ref. 8]. Objects interact by passing messages between one another. Objects are the basic building blocks of an object-oriented system. Figure 5 depicts objects.

An object models something in the real world that humans can identify with readily. Therefore, an object exists in time and space. Specifically, Booch says that an object is any of the following:

11
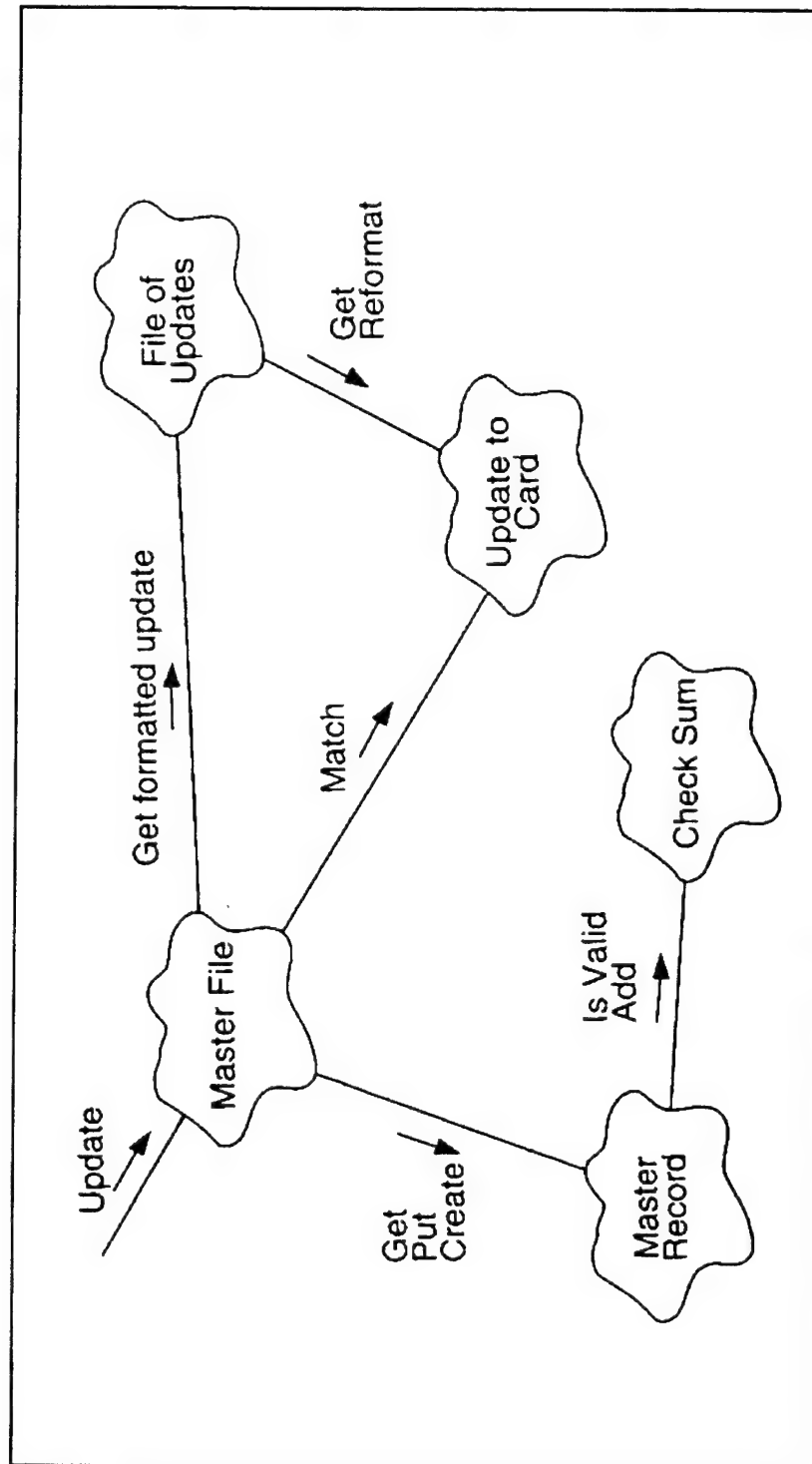
Figure 4. Object-oriented decomposition. From Ref. [8].

- A tangible and/or visible thing

- Something that may be apprehended intellectually

- Something toward which thought or action is directed [Ref. 8].
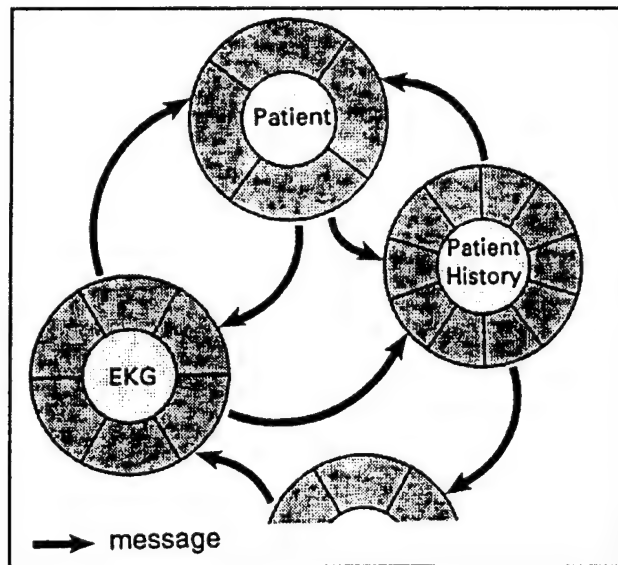
Figure 5. Object Model. From Ref. [9].

Object-oriented means that the aforementioned behavioral characteristics of objects are captured and incorporated in an object-oriented system. The object-oriented models which describe complex systems are shown in Figure 6.

a. The Physical Model

The Physical model represents a view of the system that describes the software and hardware components. It further details the architecture of the modules and processes of the system.

b. The Logical Model

The Logical model represents a view of the system that reveals essential structural details of that system to the users, while supressing some of the unnecessary details. The revealed details form the problem space.

c. The Static Model

The Static model represents a view that describes events that remain fairly constant. In other words, the state of the events depicts a scenario that is predictable in time and space.
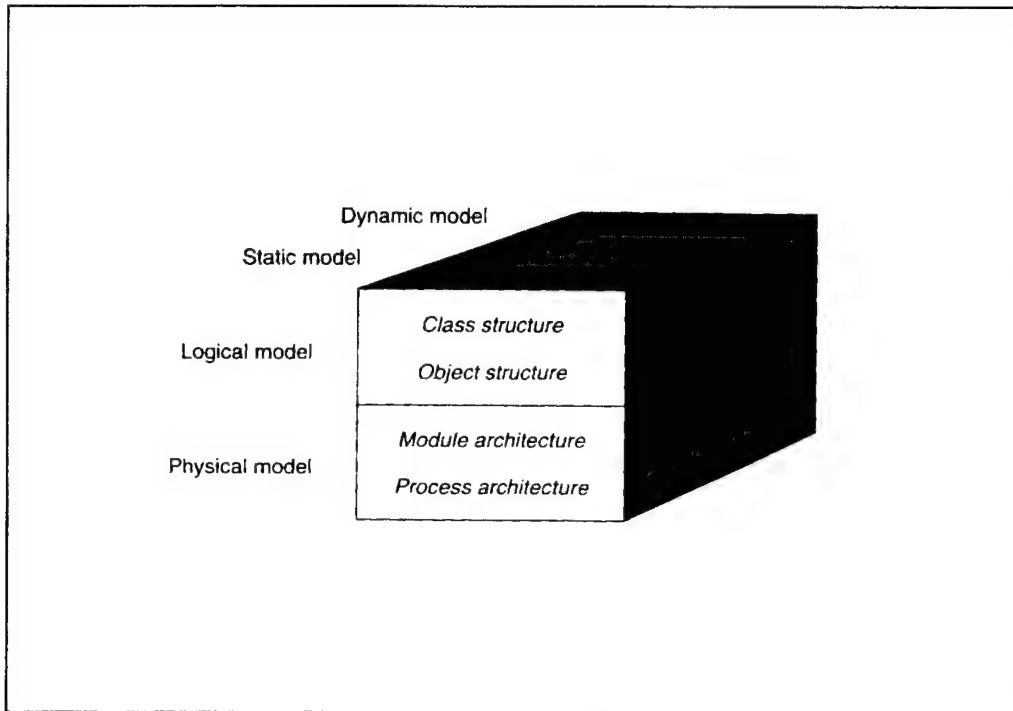
Figure 6. Object-Oriented Models. From Ref. [8].

d. The Dynamic Model

The Dynamic model represents a view that shows events that cause change or are changed themselves by other events. Examples of dynamic models are objects that send messages, objects that are created, or objects that are destroyed by some process.

In summary, the object model represents something that exists in the real world. The object-oriented model uses the four representations to create objects from a real world abstraction.

2. Object Characteristics

According to Booch, a model which bears the title, object-oriented, must possess four characteristics and may possess three others. [Ref. 8] The first four characteristics described are essential to the object model and are considered to be major elements of the object-oriented model. The last three mentioned are useful, but nonessential characteristics are minor elements.

a. Abstraction

Abstraction is that characteristic of objects that differentiates it from other objects because its conceptual boundaries are solely the pespective of the viewer. The outside of the object is the focus of abstraction. Since the outside of the object is the view observed, the object's extrinsic *behavior* or *method* serves as the interface with other objects. The object's intrinsic *attributes* or *instance variables* store the identity of the object. It is important to note that an object's extrinsic behavior (method) communicates with its own intrinsic attributes (instance variable). However, an object can only communicate with another object through its method.

The principle of abstraction is seen to be one of the keys to success in object-oriented programming, if the problem domain is broken into "good" pieces. [Ref. 9] In this context, goodness can be defined by asking, can I explain the what the behavior of this object is and how it interacts with other kinds of objects. Once the question is asked, a concrete example usually helps to answer the goodness question.

An example of the concept is a wrist watch, which is an abstraction for time. The wrist watch is an object which has a dial as an interface with the object, human. A wrist watch is a good example, because its behavior and how it interacts with human can be easily explained.

b. Encapsulation

Encapsulation is complementary to abstraction. Encapsulation is the secret about an object's behavior. In short, it is the inside view of the object. The inside view is hidden from the viewer, and indeed should be hidden to reduce complexity. In the wrist watch example, encapsulation is the process of hiding the quartz crystal, the

15

IC chip, or mini-gears that represent abstractions of the watch at lower levels. Even how the watch components work to create dial movements is shielded from view.

c. Modularity

The concept of modularity means dividing and packaging the abstractions into logical units. Once the boundaries of the abstractions have been decided, they are called modules. The modules are the discrete, cohesive building blocks in the design of a system. They are the parts of the whole, system.

d. Hierarchy

Abstractions have different levels of complexity. Depending upon whether or not the system as a whole is described or its modular components, it is clear that one can characterize the system in generalities or specifics. Hierachy is the characteristic of abstraction that can be referred to by rank ordering the abstractions on different levels. The principle of inheritance is closely related to hierachy, and it will be discussed later.

e. Typing

Booch describes typing as the prevention of the mixing of abstractions. [Ref. 8] Typing also describes the behavioral properties that a collection of abstractions share. This concept also characterizes the term *class*.

For example, in the English language, a subject and verb put together form a sentence. An English verb and a Japanese symbol (+) cannot be mixed to form a sentence. This grammatical rule is an example of strong typing. Weak typing is when conformance rules allow the interaction of differing abstractions, but the abstractions are indistinguishable. For example, the data types, *float* (floating point number) and *int* (integer) represent different abstractions, but they may be combined even though they both come from the indistinguishable abstraction, called data type. Un-Typing is the characteristic of abstractions that interact freely, even though they may not know how to communicate or respond to the interaction. For example, in some client-server architectures, an untyped programming language might allow the server to send messages to many clients even though the clients may not know how to respond. Typing, therefore defines the rules for the integration of abstractions.

16

f. Concurrency

Concurrency focuses upon the process which make objects operate simultaneously. Specifically, concurrency deals with a process called thread of control which allows independent actions to occur in a system. Often these threads of control operate from a single processor that handles a single operation. Concurrency allows multiple threads of control to occur simultaneously. As it relates to object-oriented development, concurrency allows several objects to act at the same time. Objects that dynamically operate on a single thread of control are said to be *active.*

g. Persistence

When an object's lifetime transcends the execution of a single program, that object has the characteristic of persistence. In the time and space domains, persistence allows an object to remain in the same state. Three states may occur: data may exist between various executions of a program; data may exist between various versions of a program; or data may outlive the program.[Ref. 10].

3. Classes

Classes are defined as a collection of objects which have common behaviors and structures. The shared characteristics of objects are what give them a class structure. In fact, an object is an instance of a class. In large, complex structures, the classes may be inadequate to describe system. So, classes may be divided into subclasses. The higher level class that creates the single classes is called a superclass.

It is important to discuss the relationship that exists between classes. Relationships help to further reduce the complexity of large systems and help to increase one's understanding by, perhaps, showing some commonality between the parts that define the system. In classes, three kinds of relationships may exist: a generalization-to-specialization, whole-to-part, and association.

The generalization-to-specialization relationship is usually referred to as an "is a" relationship. For example, a cruiser is kind of ship. Ship is the generalization class, and cruiser is the specialization subclass.

17

Whole-to-part relationships denote some physical or subsidiary connection between classes or "part of" relationship. For example, the gas turbine engines are a part of a cruiser. This relationship shows that gas turbine engines are part of the whole class called cruiser. Gas turbine engines are not cruisers, and could be applied to other classes of ships.

The third relationship is called association. This kind of relationship is formed by a symbiotic bond between two or more independent classes. For example, ship and ocean have an association relationship because the ship class and ocean class, although dissimilar, form a codependency that relates to travel.

Six different combinations of relationships define a class. These relationships directly support object-oriented development.

a. Association

As mentioned earlier, association brings together two independent classes and seeks to establish some kind of dependency relationship. Association is the weakest form of relationsip of all the class combinations. It is often difficult to find associations between classes. This task is primarily important at the design stage of system development because during design, the interrelations between component parts of the system are fitted together.

There are three types of associations. [Ref. 3, p. 251]. There are one-to-one relationships, like person to name. A person has one name. There are one-to-many associations, like department to employees. A department can have many employees. Many-to-many association is the third type. An example of a many-to-many association is fans to sporting events. Fans attend many events, and conversely events can have more than one fan.

b. Inheritance

Inheritance is the property whereby subclasses inherit both the structure and behavior of the superclass. Inheritance expresses the generalization-to-specialization kind of relationship of a class. The litmus test for inheritance is as follows: "given classes A and B, if A 'is not a' kind of B, then A should not be a subclass of B." [Ref. 8].

18

c. Aggregation

Aggregation deals with the whole-to-part kind of relationship. It expresses the idea that behavior and structure of objects exists when other objects exist. The same is true for classes of objects, in that if a relationship exists between two objects of differing classes, then a relationship exists between their respective classes. Aggregation differs from inheritance by the type of relationship between objects. Remember the "is a" relationship versus "part of" relationship. For example, pages may form a chapter. The relationship is that pages are part of a chapter. They coexist. Chapters, however, could not exist without pages.

d. Using

When discussing the using relationship between classes, the terms client and server must be defined. A client only acts upon other objects or classes and is also referred to as an actor. A server is only acted upon by other objects or classes. So, the using relationship is one in which a client uses the services of a server.

e. Instantiation

Instantiation is the relationship derived from the collection of characteristics of a parameterized class (one that outlines the characteristics without the specifics) to create another class. From the class, objects are created.

f. Metaclass

Metaclass defines a relationship between the top level class and all classes below. In fact, a metaclass is a class which has instances that are classes also.

So, relationships between classes are important because they help reduce the complexity of looking at the system as a whole. Once the class relationships are established, the framework for system design begins to form.

D. OBJECT-ORIENTED SUCCESS TRAITS

Two traits that characterize most successful projects in software development are the existence of a strong architectural vision and the application of incremental and iterative development processes which are well-managed. [Ref. 8] Therefore, successful object-oriented projects will have these traits.

Architectural vision means that the software archictecture has well-defined layers that are coherent and well-defined interfaces that are controlled. There is a distinct separation of the interfaces and implementations of abstractions between layers. Changes to either the interfaces or implementations can be made without violating any assumptions that the users have made. Sound architectural vision also follows the tenet of good systems engineering: keep it simple with as much commonality between behavior and mechanisms, as possible.

Systems that are built iteratively, with the basic cycle of analyze-design-evolve, of a logical way to monitor the progression of a system as it develops. This promotes re ement of the system as it reaches the deployment stage in increments.

Object-oriented systems naturally fit the iterative pattern of analyze-design-evolve since this cycle underlies the way systems are decomposed into objects and classes. Object-oriented analysis (OOA) looks at the problem domain and examines the requirements of system development from the perspective of objects and classes. Object-oriented design (OOD) uses the object-oriented model to decompose the system and use the OOA to depict the physical and logical forms of the system. Object-oriented programming (OOP) is the process of writing programs characterized as objects which inherit from a larger class, and then clustering the objects in a logical method to form classes. Object-oriented analysis, design, and programming suitably adapt to a well-managed interative and incremental process.

E. MICRO PROCESS

Successful object-oriented development begins by means of the micro process. The micro process guides the object-oriented development through a series of activities. The activities focus primarily upon the short-term activities of development. The day to day events usually involve personnel who make the tactical decisions that move the project to completion.

The micro process establishes the boundaries of the problem by first, identifying the classes and objects at a certain level of abstraction. Once bound, the problem

20

domain is examined to find solutions that reduce the complexity to lower levels of abstraction. As classes and objects are identified, the data dictionary is formed. The data dictionary defines the decomposition, contains the flow definitions and portrays the object and class interfaces. The data dictionary serves as the deliverable for the first step of the micro process, yet it is updated at each interation of the micro process.

Next, the class and object semantics are identified. In this phase, the behaviors and attributes of the classes and objects for a given level of abstraction are distinguished. The data dictionary is refined. The object specifications are also written. The formal implementation languages, like Objective C, C++, Ada or other object-oriented languages, may be used to write the interfaces for the different abstractions. Object diagrams and interaction diagrams may also be written. Scenarios may be drawn using storyboarding techniques. Once the responsibilities for the abstractions, as well as their operations are delineated, the next phase of the micro process is ready to begin.

The third phase of the micro process begins with identifying the relationships among the classes and objects. The purpose is to define the interactions and determine the separations between the objects and classes identified earlier in the process. Associations among classes and objects are analyzed. Relationships are identified by class diagrams, object diagrams, and module diagrams. During the third phase, associations are refined and specified, and collaborations are identified. When a sufficient blueprint for the class and object implementations is developed from the prior phase activities, as well as the relationship identifications in the third phase, development moves to the fourth phase.

The fourth phase is the last sequential phase of the micro process. During this phase, class and object implementations refine the abstractions to yield more innovative classes and objects. The new classes and objects lead to the next iteration in the micro process. Decisions about the physical design are part of the next sequential phase in the macro process, but the fourth phase provides more tangible abstractions for the physical model. Psuedo-code (executable) code is written and the data dictionary is updated to define the new abstractions. The structures and algorithms

21

for the design are derived. Now, the focus shifts from an outside behavioral view to an inside attribute view of the classes and objects. Once all essential interesting abstractions and their responsibilities at the higher level have been identified, the implementation takes place. This process continues in an iterative fashion until the executable code and nonexecutable models are designed.

F. MACRO PROCESS

While the micro process focuses upon the short-term, day to day activities of software development, the macro process takes a broad look at the overall development process. The macro process drives the micro process in that it guages the progress of the micro process, and tracks the cost, schedule, and performance issues of project management. The macro process monitors progress based upon five major activities.

First, the macro process examines the conceptualization of the development. During conceptualization, the core requirements of the program are established. New ideas are conceived and validated by the architect, engineer, or analyst. From conceptualization, prototypes are developed. Prototypes are not the production line deliverables, but the proof-of-concept models that validate the designer's concept. [Ref. 11] Strict adherence to development rules encumber the designer, so the rules aren't necessarily followed during the conceptualization phase of the macro process. The end product of the conceptualization phase is a formalized production process derived from the proof-of-concept.

Next, in the process is analysis. In this phase, behavior is the focus. The behavior of the system is captured to describe the problem in testable terms. All the primary behaviors of the system are examined so that essential issues are tackled. The analysis produces a functional description and risk assessment of the system. Several scenarios are planned for the system to determine its behavior. Development moves to the next phase once the primary system behaviors have been agreed upon by the development experts and end users.

22

Then, the design phase begins. The design creates the architecture for the system. The rules for designing the architecture and a description of the design are written. Planning, design, and release activities occur. The design phase is complete once the design policies are signed off and the prototype is validated through formal reviews. It is during this time that flaws in the architectural design should be discovered, so that solutions could be developed before succesive releases or production items are built.

In the fourth phase of the macro process, the implementation of the system evolves to maturity. In other words, the system is refined towards a production line item. At this stage, the item is deployed to the customer. Updates to the system bring it to the point of refinement and satisfaction for the user. The evolution phase is complete when the product is delivered according to the design specifications and user satisfaction.

Maintenance is the last phase of the macro process. During this phase, post-delivery issues are handled. Some changes and debugging of the system are performed in this phase. Additionally, upkeep and system support are handled. Resources are allocated to support the architecture, as well. The macro process ends when the bugs have been worked out and no existing production releases are necessary.

In summary, object-oriented design is a systematic approach to decomposing complex systems and then reconstructing them from simpler models. Reducing complexity is the ultimate goal of an object-oriented approach to developing systems.

III. CDSSI AS OBJECTS

A. USING OBJECTS

CDSSI uses an object-oriented database (OODB) as the source which transfers and stores the information, in real-time. Real-time, as far as the development of CDSSI is concerned, is defined as having no delay caused by the infrastructure systems that is unacceptable to any system. [Ref. 2] The process of delivering information on demand without the encumberances of enormous time delays and modifications to the external systems, is made easier with object-oriented development. It is also one of the key aspects of interoperability. This chapter discusses how CDSSI is subdivided into objects.

In Generation One of CDSSI, information transfer and storage was done by flat files and messages sent from the server to the external clients. The glue code allowed interprocess communications to occur between the Common Data Sharing System, and the connected C^2 systems. Although the information transfer was transparent to the client, the using relationship between the client and server became overloaded when the messages being passed between different clients required updates or modifications.

The use of objects in the development process primarily applies to the Generation Two architecture. One of the lessons learned from the Generation One prototype was that the use of flat files for information storage was inadequate to provide the robustness necessary to meet the changing requirements of the CDSSI. In CDSSI, the object is a unit of transferring information and is a measure of system performance in order to compensate for the shortfalls of the Generation One development.

B. CDSSI AND THE MACRO PROCESS

As discussed in Chapter II, the macro process is the base structure for the project being developed. It guides the day to day activities of the micro process.

Through incremental stages and iterative steps towards development, the macro process provides a big picture view of project management. Figure 7 depicts the macro cycle.

Conceptualization begins the macro process. During this phase, the core requirements of CDSSI have been established. The vision for the prototype was captured in these stated requirements:

- The infrastructure must be able to interface with diverse external existing systems both real and simulated without requiring changes to the external systems. The infrastructure must be able to transfer information to/from any external system.

- The infrastructure must provide a consistent information access mechanism to all using systems.

- Persistent information shall be stored by the infrastructure.

- The infrastructure shall be capable of performing in real-time.

- The infrastructure should be able to easily adapt to new systems. Adding new systems should require creation of minimal glue code interfacing software.

- The infrastructure should be designed and implemented using open systems techniques.

From the stated requirements, the prototype begins to become a more lucid concept. The next phase of the macro process begins.

The next phase of the macro process is the analysis. During this phase, the problem that the CDSSI prototype seeks to solve is defined. The problem domain is defined by the creation of an architecture that describes the C^2 systems. For this problem to be solved, the behavior of CDSSI must be examined by identifying objects. The performance measures are also identified in the process.
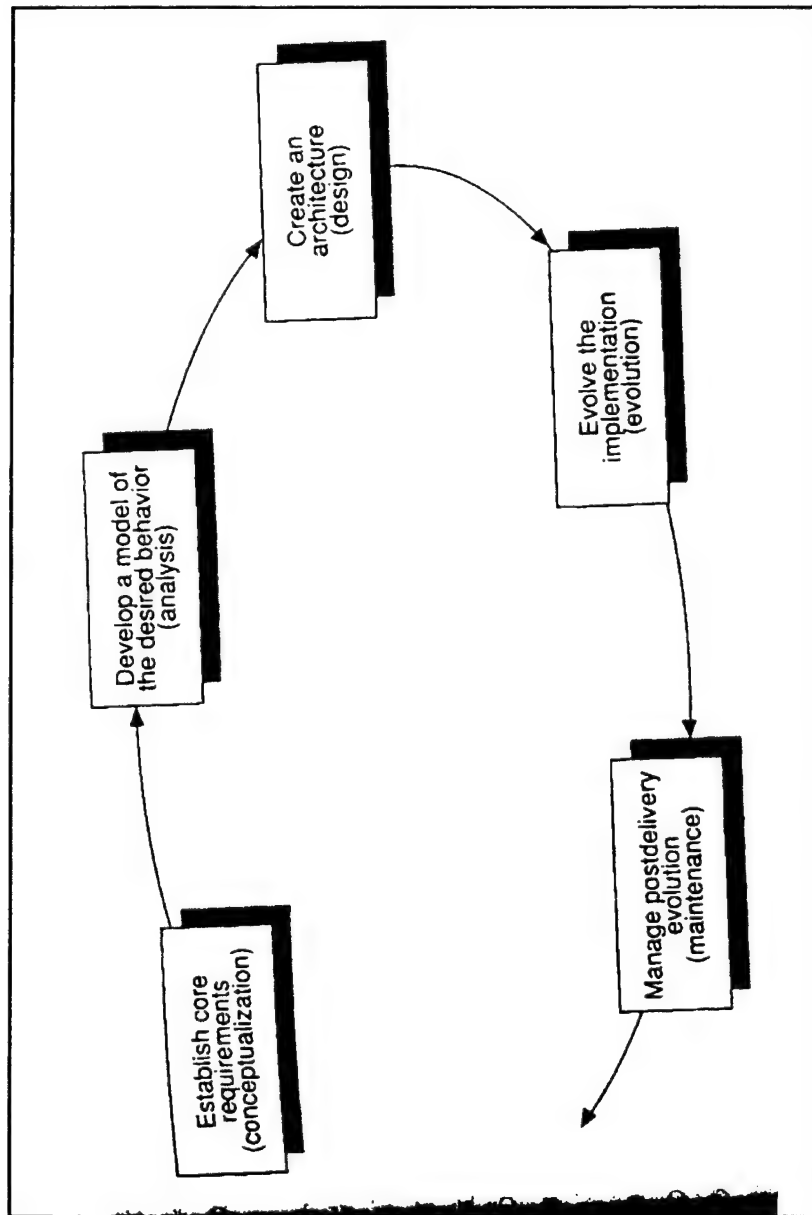
Figure 7. The Macro Process. From Ref. [8].

The objects identified in the analysis phase are the System Interface object, System Report object, Database (DB) Schema object, and Database object. These objects are the initial abstractions identified by the BESC analysts to develop the information architecture.

Form is not the issue at this point, but behavior is the focus. When external sources desire to send data to CDSSI's object-oriented database, data is transferred via the System Interface object to generate the Sensor Report object. The Sensor Report object stores itself based on the DB Schema object, and the stored data is written to the Database object. The measured overall rate of information flow through CDSSI is called throughput (Sensor Report objects per second, SRO/sec.). This event trace is described in the interaction diagram in Figure 8.
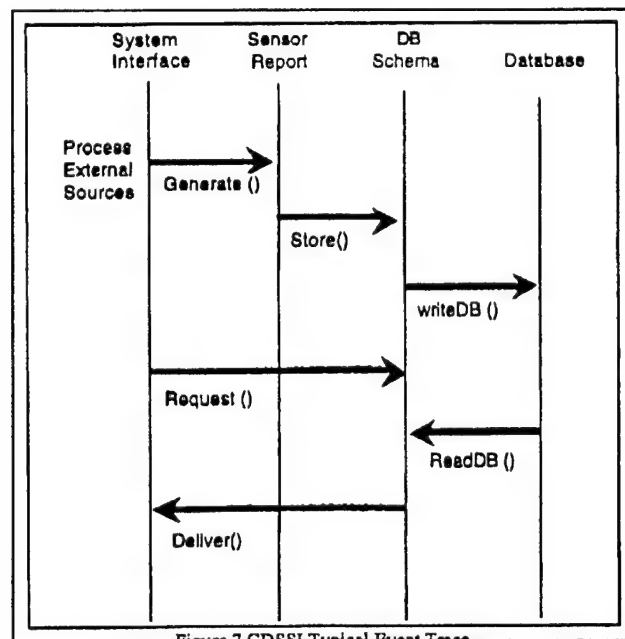
Figure 8. CDSSI Interaction Diagram. From Ref. [2].

Information retrieval is behavior that, similarly, starts with an external system that sends a request via the System Interface object. Based upon the schema, the request goes to the DB Schema object. The request is written to the Database object, and the data is read back to the DB Schema object. Data is then delivered through the System

28

Interface object to the requesting external system. The measured rate of requests serviced through CDSSI is called transaction processing rate (Sensor Report objects per transaction, SROs/transaction).

Using the micro process, the first two phases of identifying the classes and objects and identifying class and object semantics are imbedded in the macro process. The diagram shows how information may be delivered on demand. It also depicts the theory behind the concept of seamlessly delivering to external C^2 sources and interactively sharing information between sources.

The Common Data Sharing System Infrastructure is developed to make the process of information transfer easier. Diagrams help to see the process work. This chapter showed that CDSSI can use objects to help developers redefine the structure and create a useful prototype.

IV. CDSSI AS CLASSES

A. CONTINUOUS DEVELOPMENT

As seen earlier in the development methodology diagram, Figure 2, the Information Architecture depends upon inputs from both the user community and the macro/micro process. In Chapter III, the decomposition of the object-oriented CDSSI led to a description of the objects using the macro process as the broad framework and the micro process for defining the process. Continuing iterations, using the expertise of the BESC development team and user community, have amplified the process of building the Information Architecture.

Now, object and class relationships can be refined. Both class and object interfaces and implementations can also be specified. The objects that are identified in the CDSSI are:

- System Interface object
- System Report object
- Database (DB) Schema object
- Database object

Behavior scenarios are also created and identified. The behaviors that are associated with the objects are:

- Generate
- Store
- Write
- Read
- Request
- Deliver

Once the behaviors and objects are identified, the classes can be defined. Keeping in mind that the class embodies the common behavior and structure of objects, the CDSSI classes adhere to this concept.

31

1. Classes

The inheritance relationship is the primary relationship among the objects. Inheritance connotes the "is a" relationship, pertaining to the generalization-to-specialization kind. In this type of relationship, a hierarchy is established in which a lower level specialized class inherits the behavior and characteristics of the higher level generalized class.

In CDSSI, a Systems Interface class is created in which the Systems Interface object inherits the characteristics and behaviors of the C^2 systems interfaces that interact with it. A Sensor System class embodies the instances of sensor systems that provide sensor information. A Sensor Report class uses the Sensor System class to store its information to the database. The DB Schema class defines the view or window that the Sensor Report class sees through so that the correct information requested or generated by an external system is delivered.

2. Relationships

The inheritance relationships that exist between the classes and subclasses are shown below in the example CDSSI class diagram, Figure 9. Following the litmus test described in Chapter II, it is apparent that the classes have an inheritance relationship. The subclasses inherit the behavior of the superclass. For instance, the radar is a kind of sensor system, and it assumes the behavior and characteristics of the Sensor System class. Continuing in this vein, as a subclass, Sensor System inherits the behavior of the System Interface class for a particular C^2 system.

Note the using relationship that exists between the Sensor Report class and the Sensor System class. The Sensor System supplies the Sensor Report with information based upon which object (DSP, A/C radar, or Radar) given to the sensor system.

CDSSI has passed through the third phase of the micro process, identifying class and object relationships, since the abstractions of interest have been specified. A blueprint for implementing and interfacing of the objects and classes into an executable programming language now exists. On the macro level, the transition

32

between phase two (analysis) and phase three (design) has led from a description of the problem and system behavior to the creation of the physical architecture.
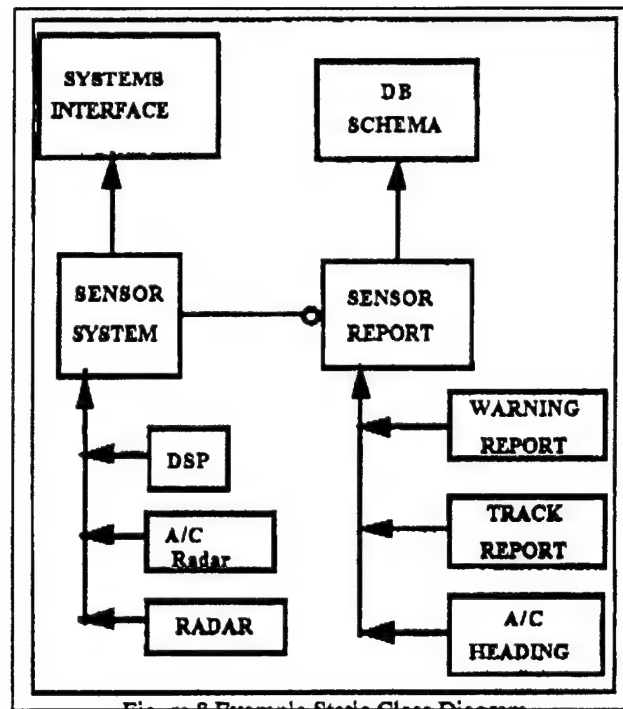
Figure 9. CDSSI Class Diagram. From Ref. [2].

This chapter discussed how identifying the system's behavior and finding relationships between the objects can provide a workable solution to system development. The embodiment of common behaviors and structures of objects is known as classes.

V. INTEROPERABILITY WITH OTHER C² SYSTEMS

A. IMPLEMENTATION AND INTERFACES

As CDSSI progresses in its the rapid prototyping development, there comes a point when the architecture must be designed to accomodate an interface between the inside view and outside view of the system. The inside view of the system which encompasses the behavior of the system is called the implementation. The outside view of the system, which allows users varying degrees of access while hiding details about object and class behavior, is called the interface.

The effective, interoperable system will have an effective implementation description, as well as an effective interface. Chapters III and IV defined the behavior of the objects and classes in CDSSI, thus specifying its implementation. The CDSSI architecture was designed in two generations. Both generations were interoperable with external clients, but were developed with differing methods for transferring information between the database and external systems. This chapter focuses upon the Generation One version of CDSSI.

B. GENERATION ONE INTEROPERABILITY

The Generation One proof-of-concept model of CDSSI showed that an architecture could be successfully developed which would allow differing C² systems to seamlessly obtain information from a common source, as well as sharing information via the same common source. Generation One CDSSI showed that both real and simulated disparate C² systems could be integrated using an infrastructure that could transfer information from system to system. The main components of CDSSI are the interprocess communications (IP) and the Common Data Sharing System (CDSS)-the most important being the CDSS.

Glue code or interfacing programs were written as IPs to bridge the connection between the CDSS and external sources. A commercial product was chosen for the IP function. Generic systems which were built to use the client-server mechanisms of CDSSI, didn't require a glue code. Most external systems that operated

with CDSSI, however, did require some glue code interface because few systems were designed to work exclusively with CDSSI. When new systems were developed, the glue code had to be changed to accomodate the new systems. Within the CDSSI, however, the formatted messages were used as primary data structures to ensure the proper size and content of messages were sent. The data structure used internally in the CDSSI required the CDSS and glue code to interpret the message. This nontrivial task becomes complicated, especially when store-retrieve-transmit-store activities were requested by multiple clients.

Often modifications occurred to the message, so several iterations of interpretations occurred before the message was properly sent to the external source. Information transfer was transparent, however, to the end user. Nonetheless, the internal controls for the message transfer needed to be refined. A standard internal infrastructure message was later created to ease the internal process control.

C. DEMONSTRATION OF GENERATION ONE

In 1993, BESC demonstrated CDSSI's capability as an infrastructure that provided interoperability between disparate systems. Demonstrations of CDSSI began as a series of smaller live demonstrations. Later, in October 1993, a large major demonstration occurred.

Players in the demonstration were located across the country. The external players, as indicated in Figure 10, were:

- A Defense Support Program, via TALON SHIELD

- A ground-based PAVE PAWS radar, in Eldorado, Texas

- The Lawrence Livermore's National Laboratory LLYNX Event Monitoring System's, Advanced Display System (ADS)

- USSPACECOM's Space Command Center, in Colorado Springs, Colorado

36

- Simulated systems: Theater Command and Control (TC^2) center, Theater Anti-air Defense (THAAD) system, Theater Missile Defense (TMD) system, and Tactical Ground-based Radar (TGBR)
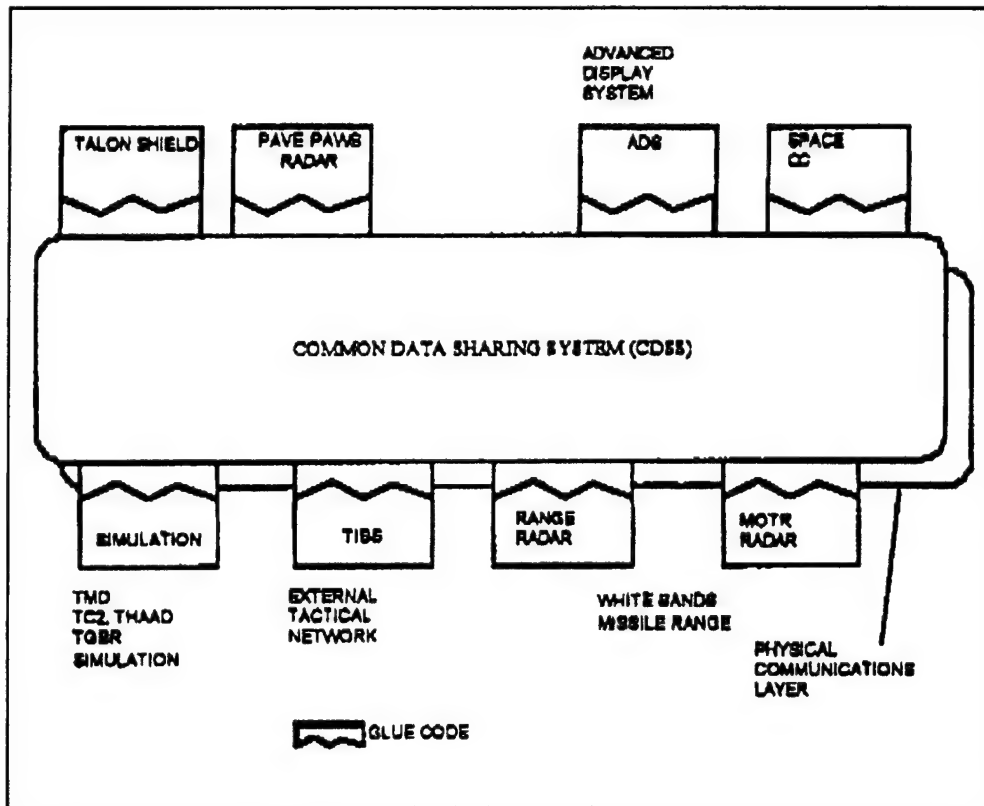
Figure 10. Demonstration of Generation One. From Ref. [2].

- The Tactical Information Broadcast System (TIBS)

- An AN/FPS-16 ground-based radar at the White Sands Missile Range (WSMR) facilities in White Sands, New Mexico

- A Mobile Tracking Radar (MOTR)

For the major demonstration, the Ballistic Missile Defense Office conducted a live flight test at the White Sands Missile Range. A theater ballistic missile (TBM) was fired from the test range in New Mexico. The BM/C^3 directorate used the opportunity to test the queuing for the CDSSI. Information was transfer and received by each of the

external systems. Once the missile was fired, inputs from TALON SHIELD were put in the system and then passed to the PAVE PAWS and MOTR radars. The initial queues provided tracking information of the TBM over the White Sands Missile Range. The result of the radars' tracking information was injected into the CDSS. The ADS displayed the information real-time in a 3-D, high resolution picture as it came to the CDSS. The queuing and tracking information from TALON SHIELD, PAVE PAWS, and the AN/FPS-16 radar was sent to the simulated Theater Command and Control center and Tactical Ground-based Radar systems. The end result was that the TC^2 successfully intercepted the live TBM.

This research, indicated that CDSSI could successfully support real and simulated command and control centers with seamless information sharing and a common tactical picture. Further evaluation of efforts to improve the Generation One's capabilities will be discussed, in Chapter VI.

VI. APPLICATION OF OBJECT-ORIENTED PROGRAMMING IN INTERFACES

A. OBJECT-ORIENTED APPLICATION

The BESC development team realized that with commercial standards and commercial off-the-shelf (COTS) technology, significant advantages could be gained in building CDSSI. The Generation One development and demonstration proved that differing command and control centers could share real-time information and observe a common graphical, 3-D presentation using a common infrastructure. After the first prototype of CDSSI was built, BESC realized that more innovative ways were required to increase the robustness and flexibility of CDSSI.

Object-oriented macro and micro processes described earlier indicated that CDSSI would lend itself well to this iterative development process. From a systems engineering point of view, object-oriented development appeared to be the best way to develop a second prototype for CDSSI. Object-oriented ways to create the next build would simplify the translation of the message transfer process and give the infrastructure the flexibility to adapt to changing requirements, as well as the addition of more advanced external C^2 systems.

Object-oriented analysis (OOA) of the problem to create a seamless architecture that would allow virtually any C^2 system to obtain information on demand within a common infrastructure appeared to be the best approach to the solution. Core requirements for CDSSI were established. CDSSI was decomposed into objects and classes. The object and class behaviors were modeled.

Object-oriented design (OOD) of CDSSI laid the framework to build the architecture. The Generation One proof-of-concept and demonstrations greatly simplified the design of the next proof-of-concept. The CDSSI object and class behavior continued to evolve, so the implementation needed a new architecture. The disparate elements of the previous architecture were addressed and successively refined to make the interface between the different systems less encumbering.

Object-oriented programming (OOP) for CDSSI intersystem processes allowed the classes and objects to be organized for economy in writing code and ease in

39

making modifications. The basic concept of organized complexity is the reason behind using OOP to CDSSI development. So, BESC incorporated these object-oriented principles to increase CDSSI's interoperability with other C^2 systems.

B. GENERATION TWO INTEROPERABILITY

Software and hardware lessons learned from Generation One led to the development of a second prototype. This prototype is the Generation Two infrastructure. Feedback from the first prototype led to changes in the CDSSI internal structure and modifications to the glue code interfacing software.

Generation Two incorporated object-oriented principles for the new build. BESC used the OOD concept to refine the objects and classes after the initial decomposition of the system. The process of information storage and retrieval was modified from the use of flat files and message to the use of an off-the-shelf object-oriented database called ObjectStore. In this type of database, the object is the unit of information transferred instead of a formatted fixed or variable-lengthed message. [Ref. 3]

In an object-oriented database (OODB), the data is not a part of the programs that update the database, but a part of the schema (user view of the database). By contrast, in the nonobject-oriented way of information transfer, the data tranferred to the database is a part of the program that updates the entire database. For example, a Track Report object would be transferred, with the inherited characteristics of the Sensor Report class, DB Schema class, and external system which reads or writes the information to the database. In CDSSI, the objects inherit the characteristics of systems that they interact with. An object, such as Sensor Report, subsequently becomes a class which distributes its properties to the objects in its class.

An OODB has three major advantages over the nonobject-oriented approach. First, in an OODB approach, the schema has a much simpler structure than the entire database. Second, security is better in an OODB because a user with a particular schema is restricted to access certain information, but not the entire database. Third, an OODB has flexibility to adapt to change since modifications to the database only require changing the views of concern and not the entire database. The Generation

Two glue code interface to external systems were simplified using object-oriented programming techniques and the OODB.

OOP efficiency reduced the latency of data transfer, increased the throughput of data, and increased the transaction processing rate. Figure 11 depicts the object-oriented Generation Two CDSSI.
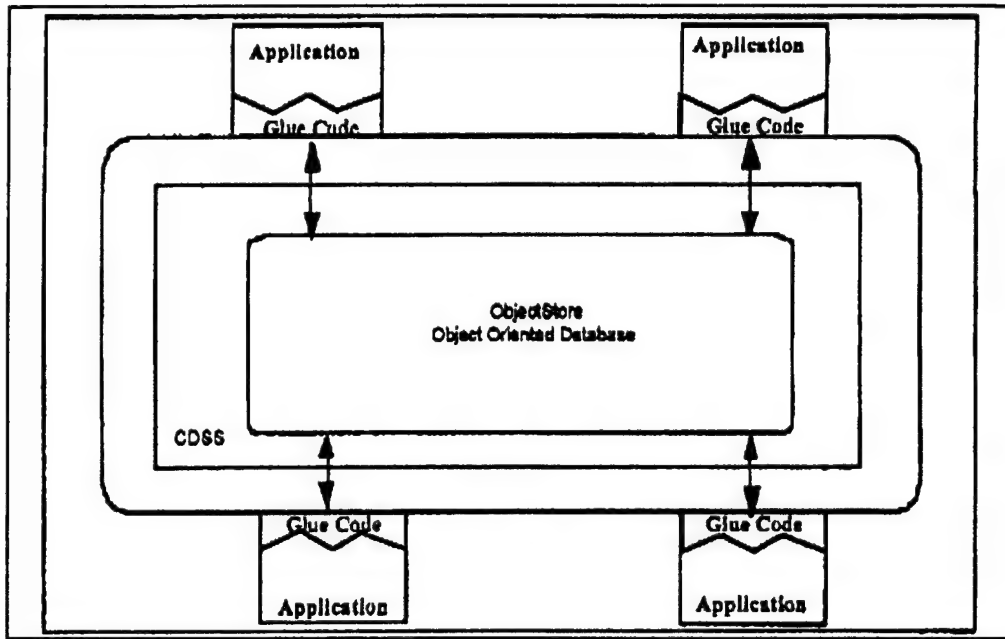
Figure 11. Generation Two Structure. From Ref. [2].

The design of the Generation Two architecture allowed new requirements to be more readily integrated into the infrastructure. The BESC team believed that Incremental design modifications in the physical, logical, static, and dynamic elements can also be made to the CDSSI. Future connections to real and simulated external C^2 systems provide an even greater opportunity for CDSSI to provide seamless interoperability for disparate systems.

C. DEMONSTRATION OF GENERATION TWO

Demonstration of the Generation Two prototype provided an opportunity to make CDSSI visible to senior military officials, commercial vendors and the BMDO development team. Obtaining high visibility is of course a double-edged sword. If the prototype works well, then wider acceptance and full production are natural by-products. However, if the prototype fails badly, it could mean the loss of future funding or cancelling of the project. The Joint Warriors Interoperability Demonstrations (JWID) 1994 was the culmination of a series of demonstrations of the Generation Two prototype.

1. The Purpose

The purposes of demonstrating CDSSI at JWID '94 were to show the following:

- Seamless interoperability among command centers at multiple echelons (JTF, JFACC, and ARSPACE);

- Enhancements/integration of command center situation awareness (S/A) and command and control (C^2) functionalities such as joint engagement, prediction of engagement results, and battle monitoring;

- BM/C^2 decision support in a joint environment;

- Seamless interoperability among tactical and strategic elements engaged in or supporting theater operations; and

- Flexibility and robustness of the infrastructure to changing theater conditions such as switching of command and control sites, communications routing, allocation of external functions, and substitutions of functions. [Ref. 1]

2. Technological Support

CDSSI used several diverse real and simulated systems to demonstate its functionality. Technological support was provided by the Electronic Systems Center (ESC) who developed the Commander's Integrated Open-system Technology Evaluator (CIOTE). The CIOTE is an external C^2 system which has applications that calculate and display: quick-alert messages, threat fans, impact error ellipses, battle monitoring data, and performance monitoring information. CIOTE is supported by TRW's Universal

Network Architecture Services (UNAS), a commercial software package that allows the integration of mixed software (Ada and C) on heterogeneous hardware platforms; internal and external message handling; screen, database and map building; and graphical user interface (GUI) tools. Originally developed for National Missile Defense command and control, CIOTE is now being adapted to CDSSI to process Tactical Missile Defense situation awareness data. [Ref. 1].

Another C^2 system which supported CDSSI was the Look-Ahead Battle Planner (LABP). Figure 12 shows the CIOTE and LAPB.

The LAPB is a decision support tool which aids the battlefield commander in the following ways:

- Identifies cities and assets at risk, as well as identifying those at risk by colored icons;

- Permits the operator to determine the best course of action to take to meet the mission objective. It also indicates when a decision could be deferred to gain insight into the enemy's decision cycle [Ref. 2];

- Projects the course of action effectiveness by predicting engagement results;

- Allocates resources between defended areas based upon a probable battle damage assessment, given the available courses of action.

Network management was also provided to support the CDSSI during JWID '94. Only the BMDO JWID network was managed by the network management system. The purpose of this system was to be able to control and test CDSSI's ability to do fault isolation, detection and circuit switching to an operable circuit. A router at the Space Support (ARSPACE) command center in Colorado Springs, Colorado was configured to allow bandwidth sharing among the different players in the demonstration.

43
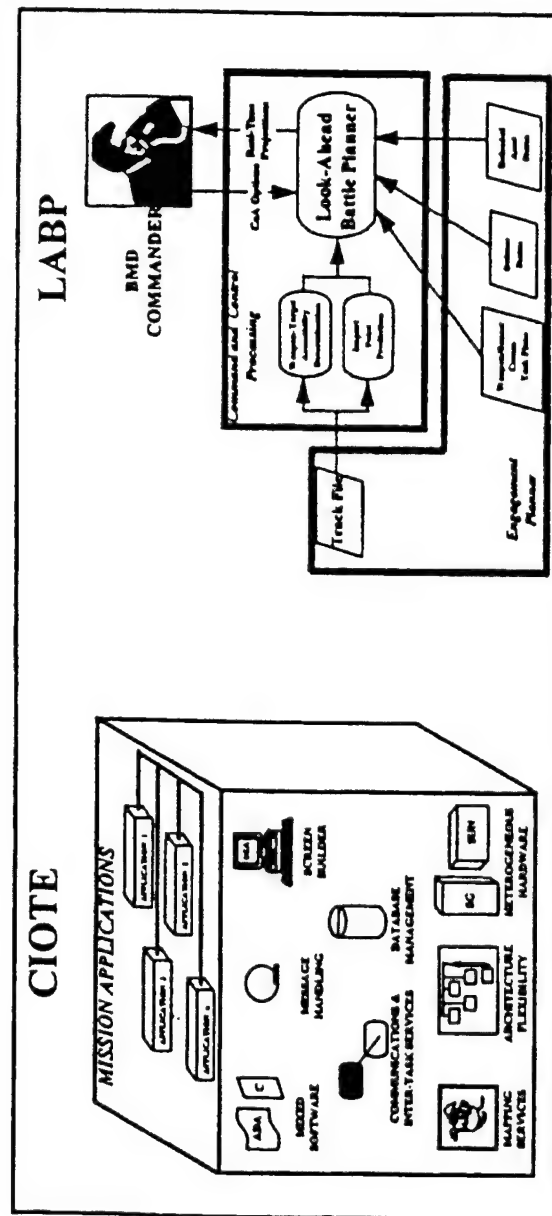
Figure 12. Look-Ahead Battle Planner and Commander's Integrated Open-System Technology Evaluator. From Ref. [1].

3. The Players

The major participants in the demonstration of CDSSI represented both real and simulated sites. The demonstration encompassed communications sites which provided various information inputs to CDSSI. The major communications sites were:

- BESC headquarters, Falcon AFB, Colorado Springs, Colorado

- ARSPACE headquarters, Colorado Springs, Colorado

- Simulated Commander Joint Task Force (CJTF) headquarters, Langley AFB, Langley, Virginia

- Simulated Joint Forces Air Component Commander (JFACC) Air Operations Center (AOC), Ft. Bragg, North Carolina

- Peterson AFB

Figure 13 shows the communications connectivity between sites. The local area network (LAN) connections and wide area network (WAN) connections are also shown.

a. Inputs

- Simulated Tactical Ballistic Missile (TBM) threat truth data

- Simulated surface radar-generated TBM tracks

- Simulated DSP sensor data

- Simulated TALON SHIELD/ALERT TBM messages

- The JTBP information generated by the Joint Maritime Collaborative Information System (JMCIS) Master Workstation

- TC^2 Battle Management/Engagement Planner Status and directives

- Look-Ahead Battle Planner generated battle recommendations

- Live TIBS and TDDS information

- Live TADIL J (LINK 16) data from JWID
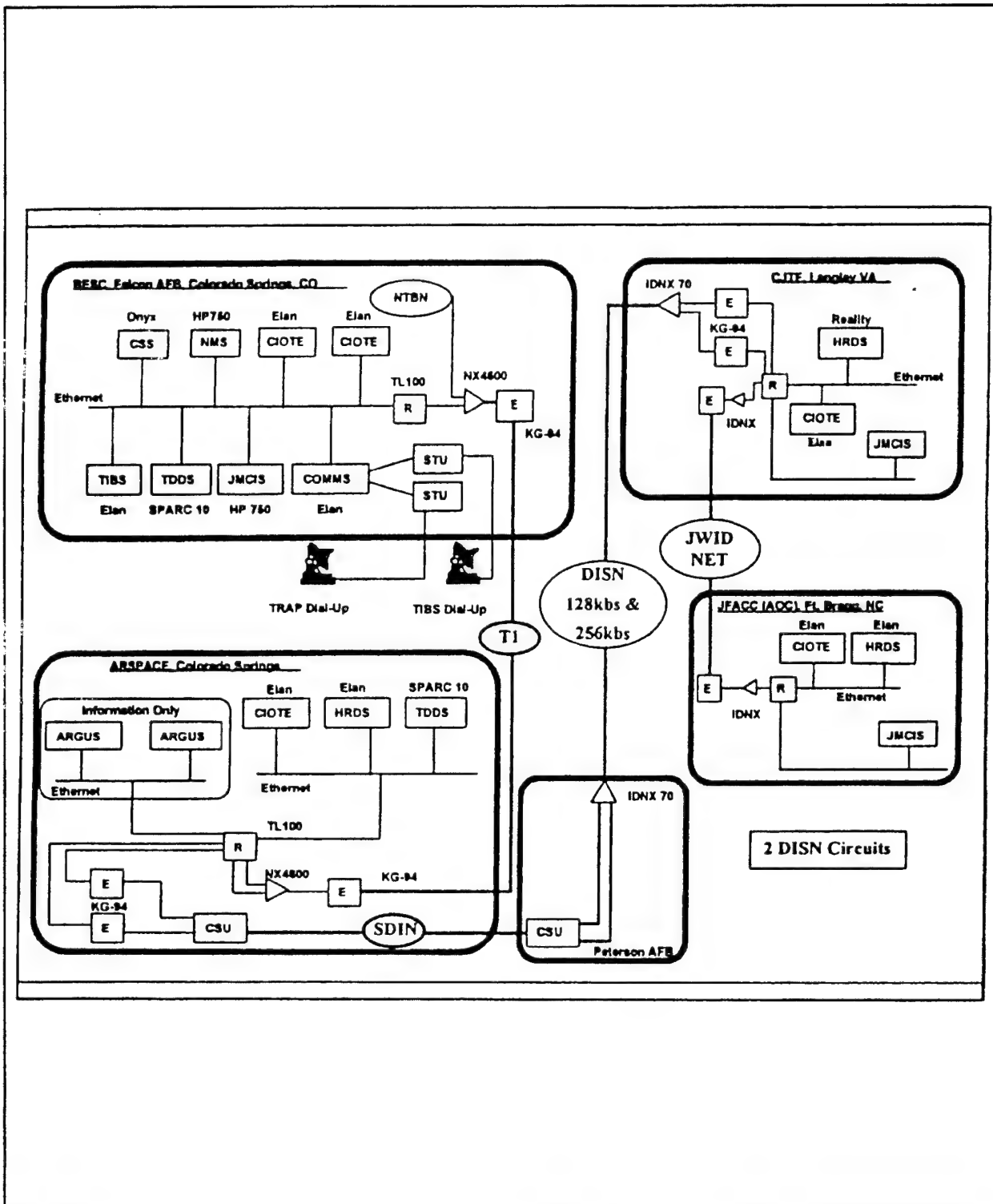
Figure 13. JWID '94 Site Communications Connectivity. From Ref. [1].

46

b. On-Site Architecture

See Table 1 for details.

c. Scenario

For the Generation Two demonstration, the basic scenario was that multiple simulated threats are launched against deployed U.S. and Coalition forces four times during two crises in JWID 94. [Ref. 1] The TBM information was injected into the CDSS by a simulated DSP via the TIBS and TRAP broadcasts. A simulated Navy Aegis platform, a simulated theater ground-based radar, and their corresponding simulated TC^2 also provided information into the CDSS. Link 16 messages were injected into the system to provide aircraft track information and authentication (identification friend or foe-IFF) information. Because of the local and wide area networks, other C^2 systems were able to see the same information on a display. Even the simulated TC^2 had access to the information. The TC^2 then simulated the intercept of the TBM. Interoperability was demonstrated by allowing the different C^2 sites to be interconnected by the network architecture, yet share information from the same object-oriented database.

By the BESC team's assessment, the second prototype of CDSSI showed that connectivity was possible. It also showed that the design of CDSSI created new connectivity as it progresses toward the first production build.

The data in Table 1 shows that several front-end architecture can be interfaced with CDSSI. Real and simulated sources operating together demonstrate the potential of CDSSI to allow disparate sources to share information from a common database. [Ref. 1] An "X" indicates that the hardware or software is on site at the location indicated. A number indicates the quantity of that item on site for the Demonstration. A blank indicates that there was no connectivity available between the hardware and/or software and the site indicated, during that particular demonstration.

47

	BESC	ARSPACE	CJTF	JFACC	PETERSON
Silicon Graphics ONYX Workstation	X				
Silicon Graphics Crimson Workstation		2	X	2	
Silicon Graphics Elan Workstation	X	2	X	2	
Silicon Graphics Reality Workstation			X		
SUN Sparc 10	X	X			
HP 750	2				
TIBS	X				
TDDS display	X				
JMCIS	X		X	X	
CIOTE	X	X	X	X	
HRDS	X	X	X	X	
CDSS	X	X	X	X	
STU-III connectivity	TRAP /TIBS				
LAN (802.3)	X	X	X	X	
WAN	T1	T1/SDIN	DISN/ JWID Net	JWID Net	SDIN/ DISN

Table 1. JWID '94On-Site Architecture

VII. PROBLEMS AND ANOMALIES

A. ROOM FOR IMPROVEMENT

With any prototype, problems will arise during its development. Good systems engineers appreciate the discovery of errors because it is much better and more cost effective to detect errors early in the process than after full production of the system. Based upon the BESC team's assessment of the major problem areas identified during design and testing, CDSSI's problems could be summarized into three categories:

- Data Structure

- Performance Measures

- Architecture Interface

B. DATA STRUCTURE

The Generation One prototype, initial system design was based upon a standard functional design model. Messages were stored, retrieved, transmitted, and stored again in the form of flat files in a client-server based achitecture. This method of transferring data was cumbersome because the data structure was inflexible. Initially the messages were both fixed and variable in length. The message included control and data information within the frames, requiring a certain amount of overhead. [Ref. 13] The data structure was designed to allow the sending and receiving systems to determine if the correct number and types of messages were being sent.

The real challenge occurred when messages being transferred were modified, deleted or created during the transmission. This meant that the message would have to be retransmitted between the client-server infrastructure and the external C^2 system several times to correctly transfer the information. This retransmission caused unnecessary internal system delays during transmission, even though the data transfer was virtually seamless to the external systems. The goal was to reduce the system complexity.

49

A minor improvement was made to the data structure. Standard internal infrastructure messages were created. So, whenever data was transferred, the standard message would carry the same bit stream to service each request (whether the data being transferred was long or short in length). This occurred, however, at the expense of more overhead being added during each transmission. Table 2 summarizes the data collected for the Generation Two demonstrations by BESC.

C. PERFORMANCE MEASURES

T E S T	Test READ WRITE	Min. Trans Time (sec)	Max. # SROs	Simultaneous writes/sec	Simultaneous reads/sec	SROs /Tran	Trans/ sec
1	1W,1R	0	8	8	7	1	15
2	1W,3R	0	3	3	27	1	6
3	1W	0	11	11	----	1	11
	Average	--	--	7.3	18.5	1	11
4	1W,1R	100	500	273	220	227.2	.939
5		100	250	154	135	142.3	1.075
6		50	250	193	171	131.1	1.391
7		50	100	76	63	47.7	1.458
	Average	--	--	174.0	147.3	137.1	1.216

Table 2. ObjectStore Performance Data for CDSSI. After Ref. [2]

The Generation Two prototype solved the flexibility issue by changing the data structure using the object-oriented model. Message transfer was examined and approached using the physical, logical, static, and dynamic models to build the infrastructure. The commercially available OODB software, ObjectStore, gave the

Generation Two prototype the flexibility to adapt to the changing requirements of modifying messages and adding new external C^2 systems with minimal design impact.

The ObjectStore software was tested in a series of demonstrations for the Generation Two prototype, including JWID '94. The Sensor Report object (SRO) became the unit of measure for system effectiveness because it had the characteristic of being transferred over a measured amount of time period. This measure allowed the BESC team to measure throughput and transaction processing rate. The overall system throughput, measured in SROs per second (SROs/sec), tracked the number of SROs that could be read or written among external systems. Throughput, in effect, measures the rate of information flow through CDSSI. The transaction processing rate, measured in transactions processed per sec (#trans/sec), tracked the number of external system requests to read or write an SRO to/from the ObjectStore database. The goal of the tests was to determine the practical rate of writing and reading SROs using the ObjectStore database. [Ref. 2] By experimental design, performance parameters measured the rate of transfer based upon:

- Writes only

- Reads only

- Read/Write combined single client

- Read/Write combined multiple clients

For example, test 2 shows that there were 0 seconds allowed per transaction using a single write and three readers, but there was a maximum of 3 SROs allowed per transaction. Test 2 reveals also that with multiple readers, 3 writes/sec and 27 reads/sec went through the system. This is because of the resource contention between readers and writers. In this case, more time was spent processing readers' requests than writer's requests. By experimental design there was only one SRO per transaction. The transaction processing rate for test 2 was 6 trans/sec. This is also equated with the throughput of 6 SROs/sec because each transaction has one SRO.

51

In comparison, test 3 had no minimum time allowed per transaction with a single writer. The maximum number of measured units through the system increased to 11 SROs. The number of writes/sec increased to 11 because processing was devoted entirely to writing. The number of SROs per transaction remained the same. However, the transaction processing rate and throughput increased to 11 per second. The baseline average throughput for all scenarios combined was also 11 SROs/sec and is considered to be poor performance.

Tests 4 through 7 used a combination of single and multiple clients, but SROs were collected and cached per transaction allowing more SROs to go through the ObjectStore database. A significant increase in performance was noticed. For example, test 4 showed that by setting a minimum time of 100 seconds per transaction, 500 SROs were available for transmission. In this single writer-single reader scenario, 273 writes and 220 reads could be processed per second. A total of 227.2 SROs per transaction request could be sent which is a significant improvement in overall throughput versus the non-caching scenario in test 1. The downfall was that the transaction processing rate was significantly reduced to .939 because more transactions caused a large drop in the number of transactions that could receive attention. This resulted in a tremendous time delay in servicing client requests, thus poor performance.

So the significant finding is that by "tuning" the ObjectStore database parameters, the delay in serving SRO requests could be minimized by collecting and caching SROs. Throughput could also be optimized. The tuning balance is to reduce the cache size, which results in a lower throughput, to gain the benefit of higher transaction service rate. The result is minimum time delays and more customers serviced. One solution would be to use algorithms or Artificial Intelligence to load balance the requests for service on the ObjectStore database. This solution bears further research.

D. ARCHITECTURE INTERFACE

The other area that presented some problem was the interface between the internal infrastructure and the external systems. The commercial software, ISIS, provided the interprocess communications services to bridge the external system processes and internal CDSS processes. The glue code or interface programs for the CDSS had to be modified to adapt from the message-based flat file conversions to the object-oriented transitions. Most external C^2 systems required a glue code interface because they weren't designed to operate with CDSSI. Fortunately, object-oriented programming techniques allowed for this transition.

VIII. CURRENT STATE-OF-THE-ART TECHNOLOGY

A. INTEROPERABLE OBJECTS

The current state-of-the art technology regarding object-oriented techniques exists mainly in the commercial sector. Many development practices are occurring commercially, and they are naturally using the object concept to build new software infrastructures. According to Ray Valdes, writer for *Dr. Dobb's Special Report* magazine, some of the current trends in software development are, "the continued evolution of object-oriented programming into the area of language-independent objects, distributed computing, and compound document technologies." [Ref. 14, p. 4]

One of the current trends in the computing industry is the design of interoperable objects. Software designers are shifting their focus away from the building better operating systems, more sophisticated languages, and more savvy application frameworks to building objects that interoperate on a broad spectrum with other commercially produced software. Interoperable objects will be the new building blocks of links between large mainframe, client-server architectures and departmental computers and personal computers across local and wide area networks.

B. DISTRIBUTED COMPUTING

The interoperable object concept follows the path of distributed computing which has been around since the 1960s. By comparison, distributed computing started using a single implementation that is distributed across basic hardware. American Airline's Easy Sabre reservation system started out using this approach by having a single application that was distributed to a host of several mainframes throughout the country. Data was distributed then. Now, graphical user interfaces (GUIs), and still to come, full-scale distributed computing environments are being developed. The standard now emerging is the Distributed Computing Environment (DCE) which comes from the Open Software Foundation (OSF). [Ref. 14, p. 5]More advanced methods allow distributed applications using object-oriented techniques. The latest object-oriented technology piggy-backed on distributed computing to where interoperable objects are used in such

standards as Microsoft's Common Object Request Broker Architecture (CORBA) and Open Linking and Embedding (OLE), as well as IBM's System Object Model (SOM).

C. COMPONENT OBJECT MODEL

Microsoft's Component Object Model (COM) is emerging as another technological standard in software design. [Ref. 15] COM is a software architecture which allows applications and systems to be built from the barrage of software components created by disparate software vendors. COM's strength lies in its ability to support the following services:

- compound documents

- cross-process interoperability

- interapplication programming

- data transfer

- data storage

- naming

- error and status reporting

- dynamic loading of components, and

- intercomponent communications (across process and network boundaries)

Basically, COM allows applications to interact with each other and other systems through the behavior or method of Component Objects. The methods serve as the interface for the applications. Differing applications have software components that are defined as objects or access points for the COM's Component Objects. COM bridges the gap between different objects by allowing the Component Objects to access the methods of other defined objects. Only the methods of the other objects are accessed not their data. Objects have a mutual interface to communicate with each other via message passing, yet one object cannot change the data of another. [Ref. 15]

D. NEXTSTEP APPLICATIONS

Another key advance in object-oriented technology is the development of NeXTSTEP applications. NeXTSTEP is both an applications program development environment and GUI which is built on three layers: the Mach operating system, the display PostScript Window server, and GUI. [Ref. 16]

1. Layers

Each of the three layers has a specific function in NeXTSTEPT architecture. The Mach operating system is a version of the UNIX system. The operating system has a UNIX micro-kernel, is based upon RISC architecture, and runs on a NeXT-based 80486 and other types of computers. The display PostScript Window server receives events, sends events to applications, and draws the applications to the screen. The GUI is presented by the NeXTSTEP Application Kit. The applications run on the Window server. The GUI is also coded using NeXTSTEP's distinctive programming language, Objective C.

2. NeXSTEP Advantages

NeXTSTEP is different from Microsoft Windows and X Windows, in that it speeds the development time of applications. Programmers can customize their company's applications for specific in-house purposes. NeXTSTEP programming also makes it easy for other applications to work together because Objective C is an object-oriented language. Applications can be applied using the same computer or across a LAN. NeXTSTEP programming also gives commonality to its applications, yet makes them easily adaptable to new applications.

3. Sample Program

A sample program using NeXTSTEP programming, that gives a user a list of options to manipulate an array of floating point numbers follows:

```
//-------------------------------------------------------------------------------------
/*
 * Sample Objective C program
 *
 * This is a sample program that manages an array of floating point numbers to
 * allow a user to perform simple calculations on the array.
```

```
 *
 *By: Calvin D. Slocumb, LT USN
 */
#import <stdlib.h>
#import <stdio.h>
#import <objc/objc.h>
#import "getfloat.h"

void main(int argc, char * argv[])
{
        int index = 0;
        int a;
        char choice = 0;
        BOOL wantToContinue = YES;
        float newNum;
        float numberList[100];
        float sum, min, max;
        //This starts a loop that is exited when the user chooses option "7".

        do {
                printf("\n\n%s%s%s%s%s%s%s%s%",
                        "1: add a number to the array?\n",
                        "2: compute the sum of array elements?\n",
                        "3: print the size ot the array?\n",
                        "4: print array?\n",
                        "5: find the minimum element?\n",
                        "6: find the maximum element?\n,
                        "7: quit?\n\n",
                        "choose one:");

                choice = getchar();

                        // The array's indices start at zero and go up. The index will
                        // always be one less than the array size.
                        // This is why the 'for' loop goes from zero to one less than
                        // the array size.
                        // The program needs an array before some operations can be
                        // performed.
                        //
                        // Case 0 will print a message to this effect.

                        if ((index == 0) && (choice != '1' && choice != '7'))
                                choice = '\0';

                switch (choice)
```

58

```c
{
case '\0':   // The array is empty and needs a value.
        printf("n\n\tThe array is empty. Please add a number first:");

        break;

case '\1':  // Add a number to the array.
        printf("\n\tenter number to add:");
        newNum = getfloat();
        numberList[index] = newNum;
        index++;

        break;

case '2':  // Calculate the sum of array values and print result.
        sum = 0;
        for(a = 0; a < index; a++)
        {

                        sum = sum + numberList[a];
        }
        printf("\n\n\tSum of elements: %f", sum);

        break;

case '3':  // Print the size of the array
        printf("\n\n\tSize of array: %d", index);

        break;

case '4':  // Print the array neatly.
        for(a = 0; a < index; a++)
        {
        printf("\n\tArray element [%d] = %f", a, numberList[a]);
        }
        break;

case '5':  // Find the minimum value and print.
        min = numberList[0];
        for(a = 1; a < index; a++)
        {
                if (numberList[a] < min)
                        min = numberList[a];
        }
```

59

```
                printf("\n\n\tMin array element value:%f", min);

                break;

        case '6':  // Find maximum value and print.
                max = numberList[0];
                for(a = 1; a < index; a++)
                {
                        if(numberList[a] > max)
                                max = numberList[a];
                }
                printf("\n\n\tMax array element value:%.2f", max);

                break;

        case '7':  // Quit
                printf("Goodbye!\n");
                wantToContinue = NO;

                break;

        default:   // Invalid choice
                printf("\nChoice invalid. Please try again:\n");

                break;

        }
        getchar(); // Consume the new character.
    } while(wantToContinue == YES);
    exit (EXIT_SUCCESS);
}
//----------------------------------------------------------------------------------------------------
```

4. Portable Distributed Objects

NeXTSTEP is developing Portable Distributed Objects (PDO) in its current state-of-the-art technology. Portable Distributed Objects is an extension of NeXT's Distributed Object (DO) architecture. Both PDO and DO allow developers of software applications to construct, operate, and maintain complex client-server applications in a heterogeneous environment. [Ref. 17, p. 58] The strength of PDOs are in there ability to extend the capability of NeXT applications to non-NeXT computers.

60

To take advantage of this new technology in computing, a developer could design new client-server applications without the hassles that normally arise from developing with remote procedure calls. Portable Distributed Objects are dynamic in the sense that they allow the developer to send messages to nonexistent or even undefined objects. If the DO server implements a new object, clients can begin using that object immediately, if it conforms to the protocol of the client system.

5. Applications in Other C^2 Areas

New technologies in object-oriented decomposition have even found there way into command and control arenas. For example, the Air Force and Navy are jointly collaborating on projects to decompose the North American Air Defense Command (NORAD) and the United States Space Command (USSPACECOM) into object and class structures. [Ref. 18] They are using an object-oriented tool, Smalltalk, to decompose NORAD and USSPACECOM into classes and objects to find the common behaviors of the two organizations. The ultimate goal is to find an effective way to consolidate NORAD and USSPACECOM into a more effective functional command.

IX. CONCLUSIONS

An issue that continues to plague the development of military C^2 systems is noninteroperability. The lack of interoperability is the result of stovepipe military C^2 systems that were designed to meet a service-specific need. It is a problem that has encumbered battlefield commanders throughout the history of warfare, as recently as the Persian Gulf War.

Since that time, however, the Ballistic Missile Defense Office has pioneered the development of an object-oriented, proof-of-concept infrastructure that has demonstrated its ability to provide seamless interoperability between disparate command and control systems. This proof-of-concept model is called Common Data Sharing System Infrastructure (CDSSI). CDSSI is being developed by a team of experts at the Battlespace Management Command, Control, and Communications (BM/C^3) Element Support Center.

BESC has shown through a series of demonstrations for a Generation One model that interoperability was achievable by using commercially available technology. The Generation One version of CDSSI also demonstrated that the use of flat files for persistent information storage was not the most effective means to achieve interoperability. Generation One testing revealed that message transfers using a fixed and variable lengthed data structure reduced the infrastructure's flexibilty to adapt to changing requirements from external systems. Consequently, the service provided to the external systems was inadequate to meet the systems functional requirements.

Object-oriented development techniques provided the flexibility and robustness needed to meet the system requirements. These techniques were applied to a Generation Two proof-of-concept infrastructure. The use of the macro and micro processes for developing the Common Data Sharing System Infrastructure provided a systematic approach to apply the two traits most common to successful project management: vision and incremental/iterative management.

The macro/micro process development practiced significantly helped the CDSSI because through iteratives building and testing of the system improvements were made

63

in the area of intersystem data transfer, increased throughput, and system flexibility. The use of the commercially available database management software, ObjectStore, gave CDSSI the flexibility needed.

Object-oriented analysis, design, and programming have shown that they provide some of the most efficient ways to reduce complexity in system design. There is, of course, no "silver-bullet" solutions to software or hardware problems. There are available tools and methodologies that have proven themselves to be effective in successfully building interoperable systems. Many of the techniques have been practiced in the past, like distributed computing. Still others are emerging, like Microsoft's Component Object Model or NeXTSTEP's Objective C and Portable Distributed Objects applications building tools. As new technologies in object-oriented systems continues to emerge, more opportunities will be available to improve CDSSI. Interoperability among disparate C^2 systems will become virtually non-existent.

The Common Data Sharing System has demonstrated its effectiveness at several demonstrations, including the Joint Warrior Interoperabiliy Demonstrations with the participation of sites throughout the country. It deserves considerable attention, simply because the infrastructure is solution-based. It also uses proven techniques for allowing interoperability to be achieved. At this point, much of the information about the Common Data Sharing System is not available because proprietary regulations prevent the sharing of data to public sources. Nonetheless, what is known is that the Common Data Sharing System Infrastructure has the potential to successfully improve a C^2 system's ability to seamlessly obtain information on demand for a battlefield commander when he needs it most. Then one command and control's most plaguing issues for battlefield commanders could be solved--interoperability.

LIST OF REFERENCES

1. "Command Center Interoperability Support to the Warfighter-BMD", *Joint Warrior Interoperability Demonstration 1994 Handbook*, pp. BM 19-29, 1994.

2. Ravenscroft, Donald L., "Interoperability of Diverse Systems Using An Object-Oriented Infrastructure", Colorado Springs, CO, 1995.

3. Pratt, Philip J. and Joseph J. Adamski, *Database Systems Management and Design*, 3rd ed., Danvers, MA:Boyd & Fraser,1994.

4. Vyssotsky, V., "On Open Systems', Unpublished Paper.1993.

5. Miller, G., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", *The Psychological Review*, vol. 63(2), p. 86, Mar. 1956.

6. Stein, J., "Object-Oriented Programming and Database Design", *Dr. Dobb's Journal of Software Tools for the Professional Programmer*, No. 137, p. 18, Mar. 1988.

7. Rentsch, T., "Object-Oriented Programming", *SIGPLAN Notices*, vol. 17(12), p. 51, Sep. 1982.

8. Booch, Grady, *Object-Oriented Analysis and Design with Applications*, 2nd ed., Redwood City, CA:Benjamin-Cummings, 1994.

9. Asbury, Stephen et al., *Introduction to NeXTSTEP for Programmers: NeXTSTEP Student Guide*, Redwood City,CA, 1995.

10. Atkinson, M., et al., "An Approach to Persistent Programming", *The Computer Journal*, vol. 26(4), p.360, 1983.

11. Nghiem, Alex Duong, *NeXTSTEP Programming: Concepts and Applications*, p. 55, Englewood Cliffs, NJ:Prentice-Hall, Inc.,1993.

12. Coakley, Thomas P., *Command and Control for War and Peace*, p. 33, Washington, D.C.:National Defense University,1992.

13. Stallings, William, *Data and Computer Communications*, 4th ed., pp. 133-138, New York, NY:Macmillan, 1994.

14. Valdés, Ray, "Introducing Interoperable Objects", *Dr. Dobb's Journal of Software Tools for the Professional Programmer: Special Report*, No. 225, pp. 4-6, Winter 1994/1995.

15. Williams, Sara and Charlie Kindel, "The Component Object Model", *Dr. Dobb's Journal of Software Tools for the Professional Programmer: Special Report*, No. 225, pp. 14-21, Winter 1994/1995.

16. Garfinkel, Simson L. and Michael K. Mahoney, *NeXTSTEP Programming Step One: Object-Oriented Applications*, New York, NY:Springer-Verlag, 1993.

17. Gentry, Dennis, "Distributed Applications and NeXT's PDO", *Dr. Dobb's Journal of Software Tools for the Professional Programmer:Special Report*, No. 225, pp. 58-61, Winter 1994/1995.

18. Jurgens, Robert B. and Terry R. Raymond, "Command Center Object-Oriented Decomposition", Proceedings of the 1994 Summer Computer Simulation Conference, 1994.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, California 93943-5101	2
3. Professor Dan Boger, Chairman, Joint C4I Systems Curriculum Naval Postgraduate School Monterey, California 93943-5002	1
4. Professor Orin Marvel, Code 39 Naval Postgraduate School Monterey, California 93943-5002	1
5. Professor Michael Sovereign, Root Hall Naval Postgraduate School Monterey, California 93943-5002	1
6. LT Calvin D. Slocumb Department Head Class 140 Surface Warfare Officers School Command 446 Cushing Road Newport, RI 02841-1209	1